**PROGRAMACION ORIENTADA A OBJETOS Y ALGORITMOS CON PYTHON**

**Programación orientada a objetos en Python**

Objetivos

● Entender cómo funciona la Programación Orientada a Objetos.

● Entender cómo medir la eficiencia temporal y espacial de nuestros algoritmos.

● Entender cómo y por qué graficar.

● Aprender a resolver problemas de búsqueda, ordenación y optimización.

<https://github.com/karlbehrensg/poo-y-algoritmos-python>

Uno de los elementos más importantes de los lenguajes de programación es la utilización de clases para organizar programas en módulos y abstracciones de datos.

Las clases se pueden utilizar de diversas maneras. Pero en este artículo hablaremos de cómo utilizarlas en el contexto de la programación orientada a objetos. La clave para entender la programación orientada a objetos es pensar en objetos como agrupaciones de datos y los métodos que operan en dichos datos.

Por ejemplo, podemos representar a una persona con propiedades como nombre, edad, género, etc. y los comportamientos de dicha persona como caminar, cantar, comer, etc. De la misma manera podemos representar unos audífonos con propiedades como su marca, tamaño, color, etc. y sus comportamientos como reproducir música, pausar y avanzar a la siguiente canción.

Puesto de otra manera, la programación orientada a objetos nos permite modelar cosas reales y concretas del mundo y sus relaciones con otros objetos.

Las ideas detrás de la programación orientada a objetos tienen más de 50 años y han sido ampliamente aceptadas y practicadas en los últimos treinta. A mediados de la década de los setenta se comenzaron a escribir artículos académicos explicando los beneficios de esta aproximación a la programación. También durante esos años se comenzaron a escribir los primeros lenguajes de programación que incorporaban estas ideas (como Smalltalk y CLU). Pero no fue hasta la llegada de Java y C++ que la idea consiguió un número importante de seguidores.

Hasta ahora, en el curso previo de esta [serie](https://platzi.com/cursos/python-cs) hemos utilizado programación orientada a objetos de manera implícita. Cuando decimos “Los objetos son las principales cosas que un programa de Python manipula. Cada objeto tiene un tipo que define qué cosas puede realizar un programa con dicho objeto,” nos estamos refiriendo a las ideas principales de la programación orientada a objetos. Hemos utilizado los tipos lista y  
diccionario, entre muchos otros, así como los métodos asociados a dichos tipos.

Así como los creadores de un lenguaje de programación solo pueden diseñar una fracción muy pequeña de todas las funciones útiles (como abs, float, type, etc.), también pueden escribir una fracción muy pequeña de los tipos útiles (int, str, dict, list, etc.). Ya sabemos los mecanismos que nos permiten crear funciones, ahora veremos los mecanismos que nos permiten crear nuevos tipos (o clases).

**Clases en Python**

Las estructuras primitivas con las que hemos trabajado hasta ahora nos permiten definir cosas sencillas, como el costo de algo, el nombre de un usuario, las veces que debe correr un bucle, etc. Sin embargo, existen ocasiones cuando necesitamos definir estructuras más complejas, por ejemplo un hotel. Podríamos utilizar dos listas: una para definir los cuartos y una segunda para definir si el cuarto se encuentra ocupado o no.

cuartos\_de\_hotel = [101, 102, 103, 201, 202, 203]

cuarto\_ocupado = [**True**, **False**, **False**, **True**, **True**, **False**]

Sin embargo, este tipo de organización rápidamente se sale de control. ¿Qué tal que quisiéramos añadir más propiedades, cómo si el cuarto ya fue aseado o no? ¿Si el cuarto tiene cama doble o sencilla? Esto nos lleva a una falta fuerte de organización y es justamente el punto que justifica la existencia de clases.

Las clases nos permiten crear nuevos tipos que contiene información arbitraria sobre un objeto. En el caso del hotel, podríamos crear dos clases Hotel() y Cuarto() que nos permitiera dar seguimiento a las propiedades como número de cuartos, ocupación, aseo, tipo de habitación, etc.

Es importante resaltar que las clases solo proveen estructura. Son un molde con el cual podemos construir objetos específicos. La clase señala las propiedades que los hoteles que modelemos tendrán, pero no es ningún hotel específico. Para eso necesitamos las instancias.

**Instancias**

Mientras que las clases proveen la estructura, las instancias son los objetos reales que creamos en nuestro programa: un hotel llamado PlatziHotel o Hilton. Otra forma de pensarlo es que las clases son como un formulario y una vez que se llena cada copia del formulario tenemos las instancias que pertenecen a dicha clase. Cada copia puede tener datos distintos, al igual que cada instancia es distinta de las demás (aunque todas pertenecen a una misma clase).

Para definir una clase, simplemente utilizamos el *keyword* class. Por ejemplo:

**class** **Hotel**:

**pass**

Una vez que tenemos una clase llamada Hotel podemos generar una instancia llamando al constructor de la clase.

hotel = Hotel()

**Atributos de la instancia**

Todas las clases crean objetos y todos los objetos tienen atributos. Utilizamos el método especial \_\_init\_\_ para definir el estado inicial de nuestra instancia. Recibe como primer parámetro obligatorio self (que es simplemente una referencia a la instancia).

**class** **Hotel**:

**def** **\_\_init\_\_**(self, numero\_maximo\_de\_huespedes, lugares\_de\_estacionamiento):

self.numero\_maximo\_de\_huespedes = numero\_maximo\_de\_huespedes

self.lugares\_de\_estacionamiento = lugares\_de\_estacionamiento

self.huespedes = 0

hotel = Hotel(numero\_maximo\_de\_huespedes=50, lugares\_de\_estacionamiento=20)

print(hotel.lugares\_de\_estacionamiento) # 20

**Métodos de instancia**

Mientras que los atributos de la instancia describen lo que representa el objeto, los métodos de instancia nos indican qué podemos hacer con las instancias de dicha clase y normalmente operan en los mencionados atributos. Los métodos son equivalentes a funciones dentro de la definición de la clase, pero todos reciben self como primer argumento.

**class** **Hotel**:

...

**def** **anadir\_huespedes**(self, cantidad\_de\_huespedes):

self.huespedes += cantidad\_de\_huespedes

**def** **checkout**(self, cantidad\_de\_huespedes):

self.huespedes -= cantidad\_de\_huespedes

**def** **ocupacion\_total**(self):

**return** self.huespedes

hotel = Hotel(50, 20)

hotel.anadir\_huespedes(3)

hotel.checkout(1)

hotel.ocupacion\_total() # 2

Ahora que ya sabemos qué son las clases y cómo las podemos utilizar en nuestros programas, platícanos qué clases te serían útiles para modelar en uno de tus programas.

**Tipos de datos abstractos y clases, Instancias**

**Abstractos**  
En Python todo es un objeto y tiene un tipo, esto significa que todo lo que hacemos en nuestro programa tiene una representación en memoria, los datos y el comportamiento se puede encapsular en un objeto.

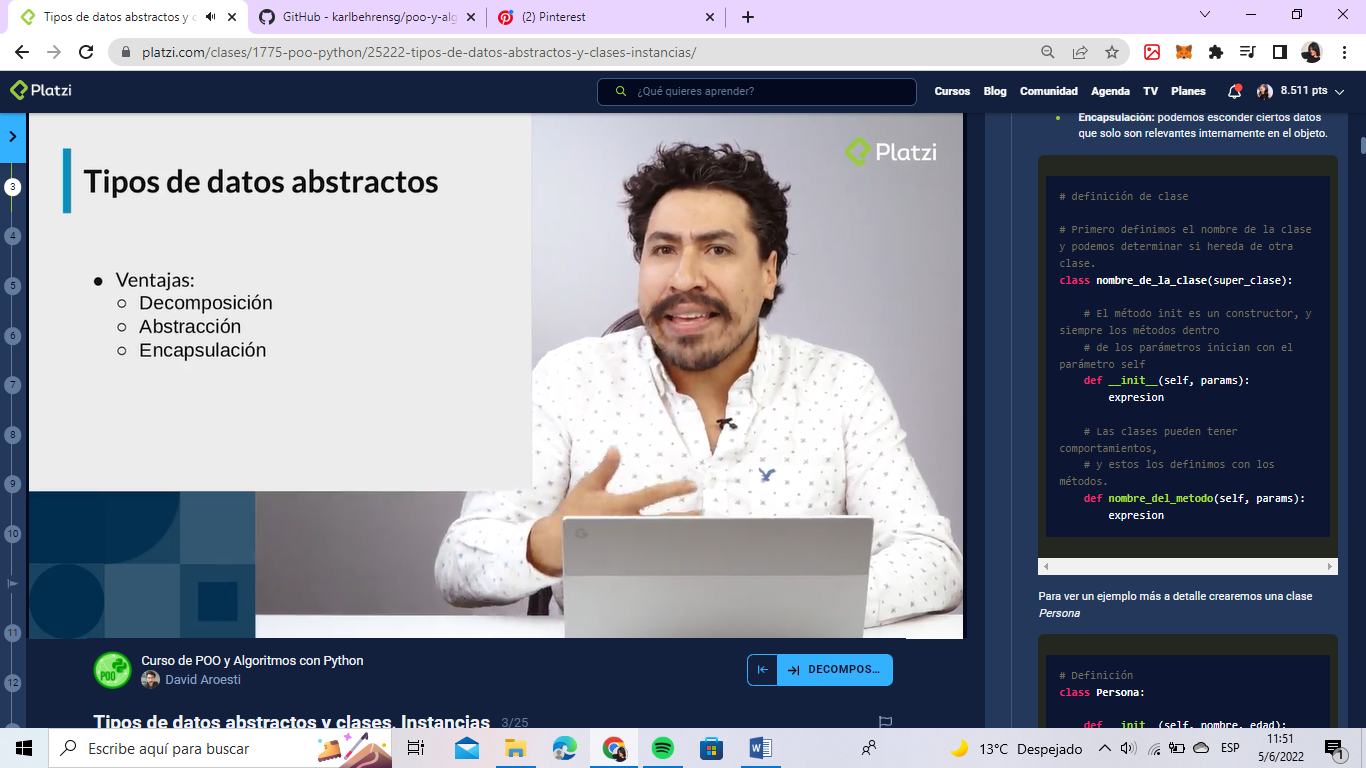
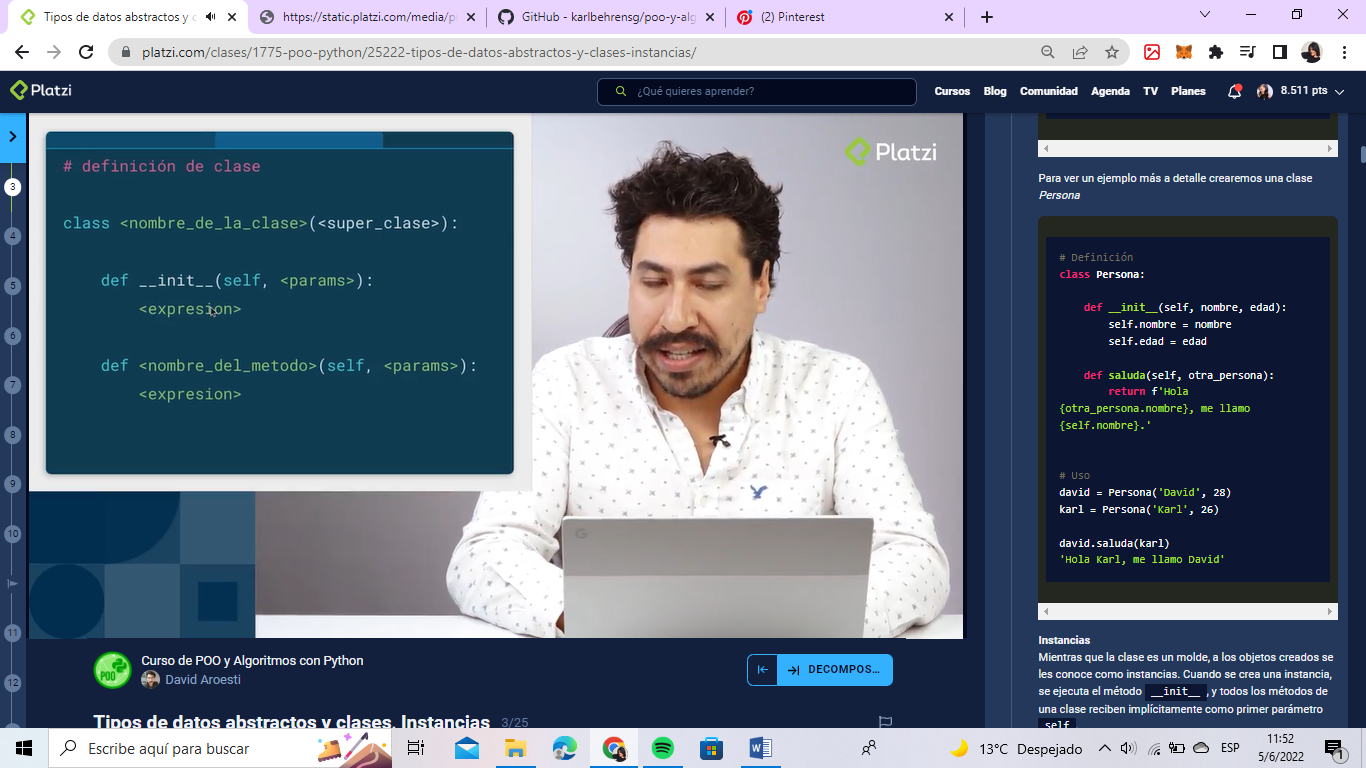
Los tipos nos permiten modelar y manipular el mundo a través de la programación.

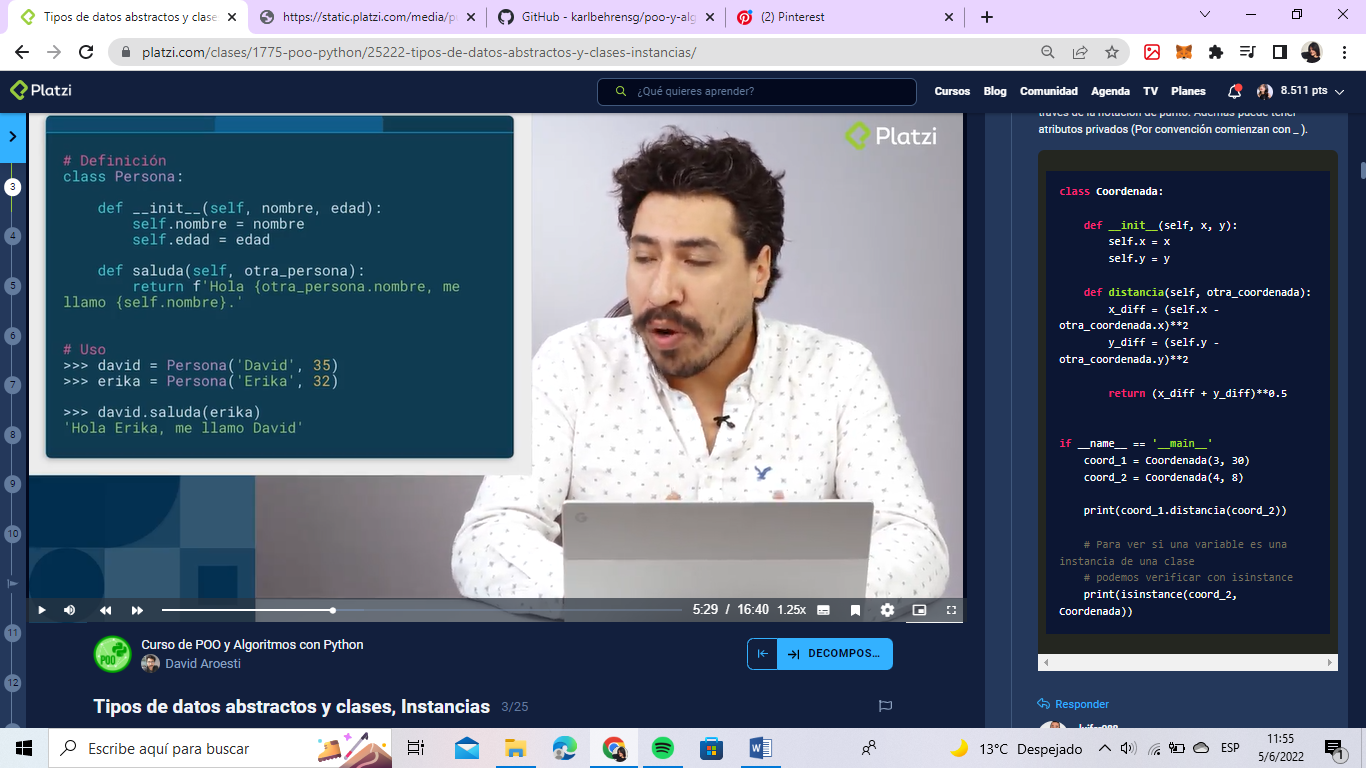
Las formas de interactuar con un objeto:

* Creación
* Manipulación
* Destrucción

Cuando trabajamos con programación orientada a objetos tenemos varias ventajas:

* **De composición:** podemos estructurarlos en objetos más pequeños.
* **Abstracción:** no nos preocupamos el funcionamiento del proceso de su comportamiento.
* **Encapsulación:** podemos esconder ciertos datos que solo son relevantes internamente en el objeto.



**Instancias**

Mientras que la clase es un molde, a los objetos creados se les conoce como instancias. Cuando se crea una instancia, se ejecuta el método \_\_init\_\_, y todos los métodos de una clase reciben implícitamente como primer parámetro self.

Los atributos de clase nos permiten:

* Representar datos.
* Procedimientos para interactuar con los mismos (métodos).
* Mecanismos para esconder la representación.

Para acceder a los atributos de estos objetos se hace a través de la notación de punto. Además puede tener atributos privados (Por convención comienzan con \_ ).

**Decomposición**

La **decomposición** es un concepto simple pero poderoso que implica:

* Dividir un problema en problemas más pequeños.
* Las clases permiten crear mayores abstracciones en forma de componentes.
* Cada clase se encarga de una parte del problema y el programa se vuelve más fácil de mantener.

Para realizar un ejemplo de decomposición modelaremos un automóvil.

**Abstracción**

La **abstracción** es:

* Enfocarnos en la información relevante.
* Separar la información central de los detalles secundarios.
* Podemos utilizar variables y métodos (privados o públicos).

**Funciones: base de los decoradores**

El concepto de decorador en Python es algo que podríamos ubicar en un nivel “intermedio” en el manejo del lenguaje, por lo que es buena idea que tengas una base sólida, sobre todo en cuanto a [funciones](https://platzi.com/clases/1937-python/29635-aprendiendo-a-no-repetir-codigo-con-funciones/) al momento de profundizar e implementarlas.

Los decoradores son una forma sencilla de llamar [funciones de orden mayor](https://platzi.com/clases/1826-java-funcional/26224-funciones-de-orden-mayor/), es decir, funciones que toman otra función como parámetro y/o retornan otra función como resultado. De esta forma un decorador añade capacidades a una función sin modificarla.

Un ejemplo de esto son las llantas de un automóvil. Si les colocas cadenas para la nieve, el automóvil aún puede andar y además extiende su funcionalidad para conducirse en otros terrenos.

**Recordando sobre funciones**

Antes de abordar el tema de decoradores haremos un pequeño repaso por las funciones, las cuales retornan un valor ante la entrada de un argumento.

Analicemos este sencillo ejemplo donde una función que multiplica un número se eleva a la tercera potencia:

def elevar\_cubo(numero):

return numero \* numero \* numero

Si damos como argumento el número 3, entonces tendremos como salida el número 27 al ejecutar la función:

>>> elevar\_cubo(3)

27

**Funciones como objetos de primera-clase**

Otro concepto importante a tener en cuenta es que en Python las funciones son objetos de primera-clase, es decir, que pueden ser pasados y utilizados como argumentos al igual que cualquier otro objeto (strings, enteros, flotantes, listas, etc.).

Veamos un ejemplo donde definimos 3 diferentes funciones que utilizaremos de manera conjunta:

def presentarse(nombre):

return f"Me llamo {nombre}"

def estudiemos\_juntos(nombre):

return f"¡Hey {nombre}, aprendamos Python!"

def consume\_funciones(funcion\_entrante):

return funcion\_entrante("David")

Las primeras dos funciones son obvias en su resultado, donde nos mostrarán un mensaje con el valor de la variable nombre. La tercera función puede ser más compleja de predecir, ya que toma otra función como entrada. Veamos que pasa cuando colocamos una función como atributo:

>>> consume\_funciones(presentarse)

'Me llamo David'

>>> consume\_funciones(estudiemos\_juntos)

'¡Hey David, aprendamos Python!'

Pongamos atención en cómo la función consume\_funciones() se escribe con paréntesis para ser ejecutada, mientras que la función presentarse y estudiemos\_juntos solo hace referencia a estas.

**Funciones anidadas**

Al igual que los condicionales y bucles también puedes colocar funciones dentro de otra función.

Tómate un minuto para analizar el siguiente código e inferir cuál será el resultado de salida:

def funcion\_mayor():

print("Esta es una función mayor y su mensaje de salida.")

def librerias():

print("Algunas librerías de Python son: Scikit-learn, NumPy y TensorFlow.")

def frameworks():

print("Algunos frameworks de Python son: Django, Dash y Flask.")

frameworks()

librerias()

Si llamamos a la función funcion\_mayor tendremos la siguiente salida:

>>> funcion\_mayor()

Esta es una función mayor y su mensaje de salida.

Algunos frameworks de Python son: Django, Dash y Flask.

Algunas librerías de Python son: Scikit-learn, NumPy y TensorFlow.

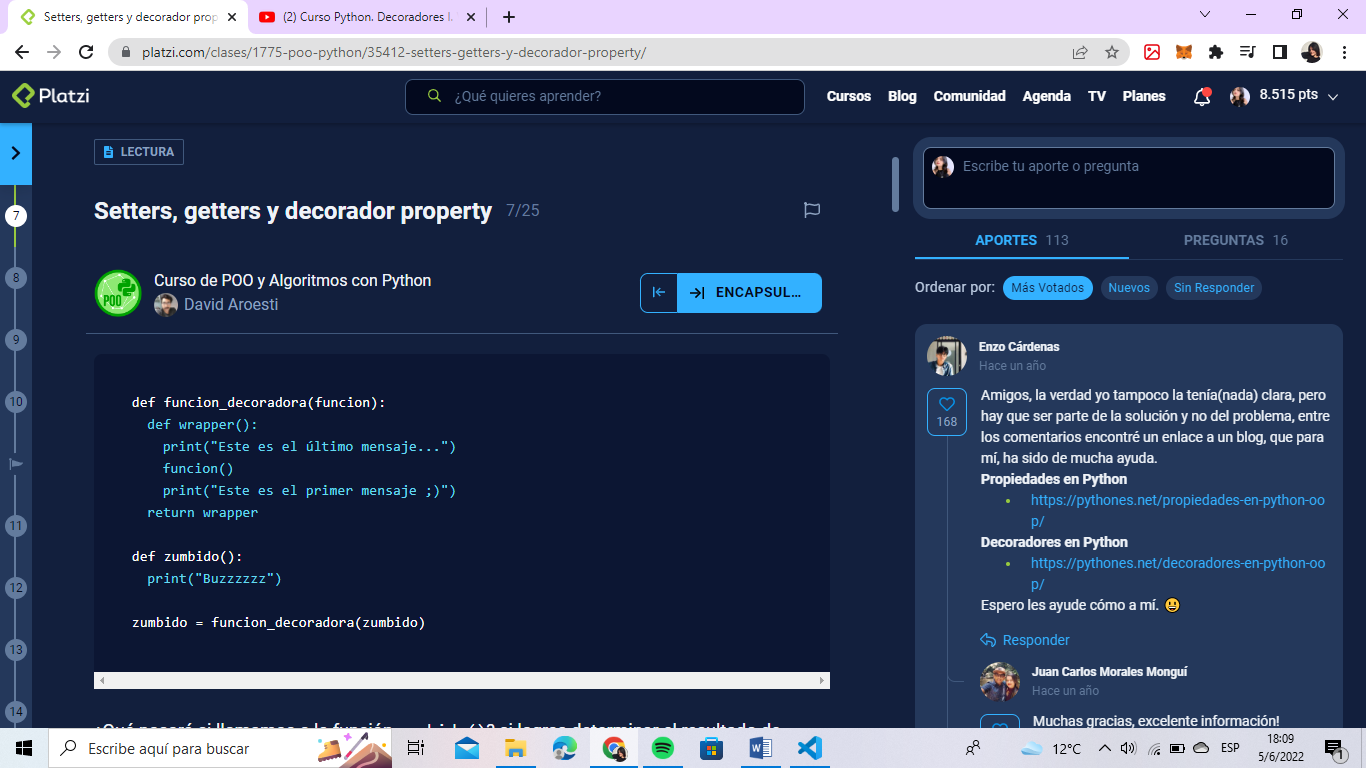
Debemos considerar que las funciones anidadas dentro de funcion\_mayor no se ejecutan hasta que se llama a esta primera, siendo muestra del [scope o alcance](https://platzi.com/clases/1764-python-cs/25241-scope-o-alcance/" \t "_blank) de las funciones. Si las llamamos obtendremos un error

**Setters, getters y decorador property**

En este punto estamos comenzando a utilizar conceptos en Python que comienzan a ser más avanzados, por lo que es normal que puedan parecerte complejos o difíciles de asimilar, así que te animo a que los repases un par de veces.

Puedes tener la tranquilidad de que si bien, al inicio no los implementas en su totalidad, podrás seguir avanzando en el curso y poco a poco incorporarlos a tus proyectos donde lo más importante es que sepas que cuentas con estas herramientas.

**Entendiendo el concepto de *decorador***

Antes de comenzar me gustaría que analices el siguiente código:

¿Qué pasará si llamamos a la función zumbido()? si logras determinar el resultado de salida o entenderlo con detalle, entonces podemos seguir adelante.

Lo que sucede es lo siguiente:

>>> zumbido()

Este es el último mensaje...

Buzzzzzz

Este es el primer mensaje ;)

Si no diste con el resultado no te preocupes, solo hay que analizarlo con detalle y el truco está en la línea zumbido = funcion\_decoradora(zumbido). Sucede que la función wrapper() recibió la la función zumbido() cómo parámetro y coloca su salida entre los otros dos *prints*.

Todo lo que sucede se conoce en programación como *metaprogramación* (metaprogramming), ya que una parte del programa trata de modificar a otra durante el tiempo de compilación. En tanto un decorador básicamente toma una función, le añade alguna funcionalidad y la retorna.

**Mejorando la sintaxis**

Definitivamente la forma en que decoramos la función es complejo, pero afortunadamente Python lo tiene en cuenta y podemos utilizar decoradores con el símbolo @. Volviendo al mismo ejemplo de funcion\_decoradora(), podemos simplificarlo así:

@funcion\_decoradora

def zumbido():

print("Buzzzzzz")

En solo tres líneas de código tenemos el mismo resultado que escribir zumbido = funcion\_decoradora(zumbido).

**¿Qué son *getters* y *setters*?**

A diferencia de otros lenguajes de programación, en Python los *getters* y *setters* tienen el objetivo de asegurar el encapsulamiento de datos. Cómo habrás visto, si declaramos una variable *privada* en Python al colocar un guión bajo al inicio de esta (\_) y normalmente son utilizados para: añadir lógica de validación al momento de obtener y definir un valor y, para evitar el acceso directo al campo de una clase.

La realidad es que en Python no existen variables netamente privadas, pues aunque se declaren con un guión bajo podemos seguir accediendo a estas. En Programación Orientada a Objetos esto es peligroso, pues podemos alterar el método de alguna clase y tener efectos colaterales que afecten la lógica de nuestra aplicación.

**Clases sin *getters* y *setters***

Veamos un ejemplo con una clase que almacena un dato de distancia recorrida en millas (mi) y lo convierte a kilómetros (km):

class Millas:

def \_\_init\_\_(self, distancia = 0):

self.distancia = distancia def convertir\_a\_kilometros(self): return (self.distancia \* 1.609344)

Ahora creemos un objeto que haga referencia a un viaje:

**# Creamos un nuevo objeto**

avion = Millas()

**# Indicamos la distancia**

avion.distancia = 200

**# Obtenemos el atributo distancia**

>>> print(avion.distancia) 200

**# Obtenemos el método convertir\_a\_kilometros**

>>> print(avion.convertir\_a\_kilometros()) 321.8688

**Utilizando *getters* y *setters***

Incluyamos un par de métodos para obtener la distancia y otro para que no acepte valores inferiores a cero, pues no tendría sentido que un vehículo recorra una distancia negativa. Estos son métodos *getters* y *setters*:

class Millas:

def \_\_init\_\_(self, distancia = 0):

self.distancia = distancia def convertir\_a\_kilometros(self): return (self.distancia \* 1.609344)

# Método getter def obtener\_distancia(self):

return self.\_distancia

# Método setter def definir\_distancia(self, valor):

if valor < 0: raise ValueError("No es posible convertir distancias menores a 0.")

self.\_distancia = valor

El método *getter* obtendrá el valor de la distancia que y el método *setter* se encargará de añadir una restricción. También debemos notar cómo distancia fue reemplazado por \_distancia, denotando que es una variable privada.

Si probamos nuestro código funcionará, la desventaja es que cualquier aplicación que hayamos creado con una base similar deberá ser actualizado. Esto no es nada escalable si tenemos cientos o miles de líneas de código.

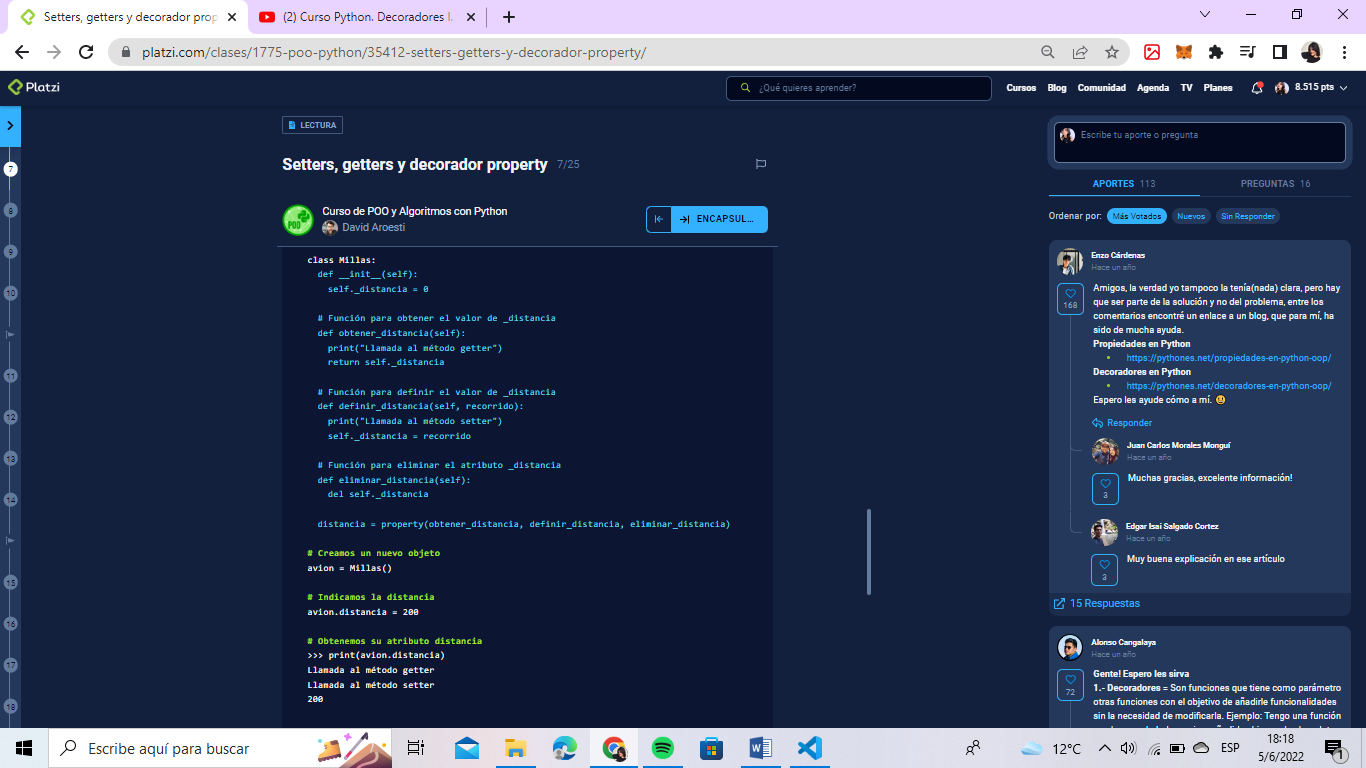
**Función property()**

Esta función está incluida en Python, en particular crea y retorna la propiedad de un objeto. La propiedad de un objeto posee los métodos getter(), setter() y del().

En tanto la función tiene cuatro atributos: property(fget, fset, fdel, fdoc) :

* fget : trae el valor de un atributo.
* fset : define el valor de un atributo.
* fdel : elimina el valor de un atributo.
* fdoc : crea un *docstring* por atributo.

Veamos un ejemplo del mismo caso implementando la función property() :

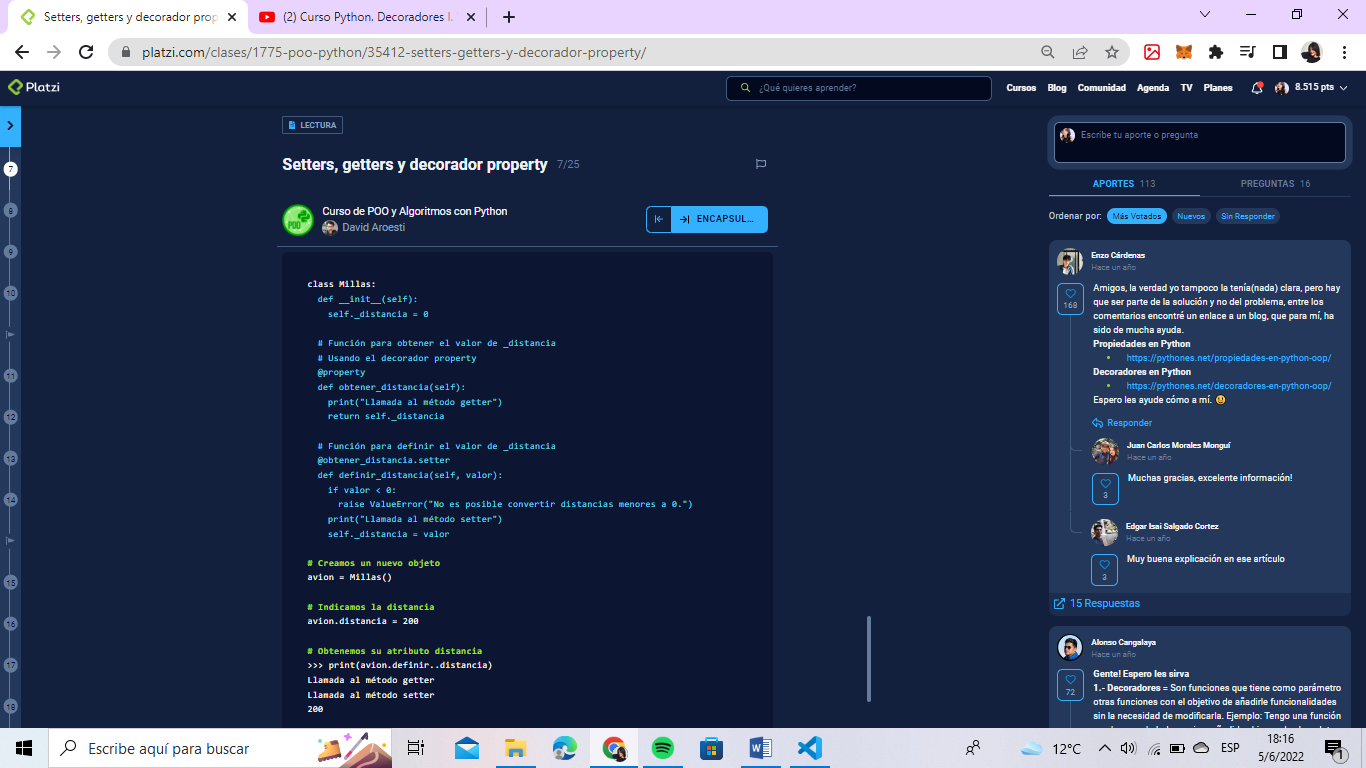


Aunque en este ejemplo hay una sola llamada a print, tenemos tres líneas como salida pues esta llama a los primeros dos métodos. Por lo que la propiedad distancia es una propiedad de objeto que ayuda a mantener el acceso de forma privada.

**Decorador @property**

Este decorador es uno de varios con los que ya cuenta Python, el cual nos permite utilizar *getters* y *setters* para hacer más fácil la implementación de la programación orientada a objetos en Python cambiando los métodos o atributos de las clases de forma que no modifiquemos el código.

Pero mejor veamos un ejemplo en acción:



De esta manera usamos el decorador @property para utilizar *getters* y *setters* de una forma más prolija e incluimos una nueva funcionalidad a nuestro método definir\_distancia() , al mismo tiempo protegemos el acceso a nuestras variables privadas y cumplimos con el principio de encapsulación.

**Propiedades en Python**

* <https://pythones.net/propiedades-en-python-oop/>

**Decoradores en Python**

* <https://pythones.net/decoradores-en-python-oop/>

<https://pythones.net/propiedades-en-python-oop/>

1. La **encapsulación** se refiere a impedir el acceso a determinados métodos y atributos de los objetos estableciendo así qué puede utilizarse desde fuera de la clase.
2. En Python **no existen los modificadores de acceso**, y lo que se suele hacer es que el acceso a una variable o función viene determinado por su nombre:  
   ***<variable> \*\* protected  
   \_\_<variable> public (note que python name mangling para evitar acceder a la variable, pero realmente sólo renombre la variable por  
   \*\**<nombre\_clase>\_\_<variable>**)
3. Entonces para qué los **getter** y **setters**. En ocasiones también puede suceder que queramos permitir el acceso a algún atributo de nuestro objeto, pero que este se produzca de forma controlada.  
   La función integrada **property()** nos permitirá interceptar la escritura, lectura, borrado de los atributos.

**Herencia**

* Permite modelar una jerarquía de clases
* Permite compartir comportamiento común en la jerarquía
* Al padre se lo conoce como superclase y al hijo como subclase

**Polimorfismo**

* La habilidad de tomar varias formas
* En Python, nos permite cambiar el comportamiento de una superclase para adaptar a la subclase

**Introducción a la complejidad algorítmica**

La complejidad algorítmica nos permite comparar la eficiencia de 2 algoritmos, esto a su vez va a predecir el tiempo que va a tomar resolver un problema. No solamente podemos analizar la complejidad desde la perspectiva temporal, también la podemos hacer desde la espacial, como por ejemplo cuanto espacio en memoria necesitamos.

La complejidad algorítmica temporal la podemos definir como T(n) el cual determinara el tiempo que demora en resolver nuestro algoritmo.

Aproximaciones

¿Cómo podríamos aplicar nuestra función T(n)?

Cronometrar el tiempo en el que corre un algoritmo. Sin embargo no es una buena forma de medir los algoritmos, ya que no se puede predecir cuanto demorara a medida que crecen nuestros pasos.

Contar los pasos con una medida abstracta de operación. Nos puede acercar a una medición ideal, sin embargo varía mucho de algoritmo en algoritmo y a medida que crece nuestro dataset existen muchos términos que llegan a ser irrelevantes.

Contar los pasos conforme nos aproximamos al infinito pero con una medida asintótica.

**Medición temporal**

Para una realizar una medida temporal simplemente calculamos la diferencia del tiempo previo y posterior de la ejecución del algoritmo.

**Conteo abstracto de operación**

Con esta técnica contamos los pasos que realiza nuestro algoritmo. En el siguiente ejemplo respuesta tendrá los números de pasos que realiza nuestro código al ejecutar.

**Notacion asintótica**

**Big O notation.**

**Crecimiento asintótico**

No importan las variaciones pequeñas.

El enfoque se centra en lo que pasa conforme el tamaño del problema se acerca al infinito

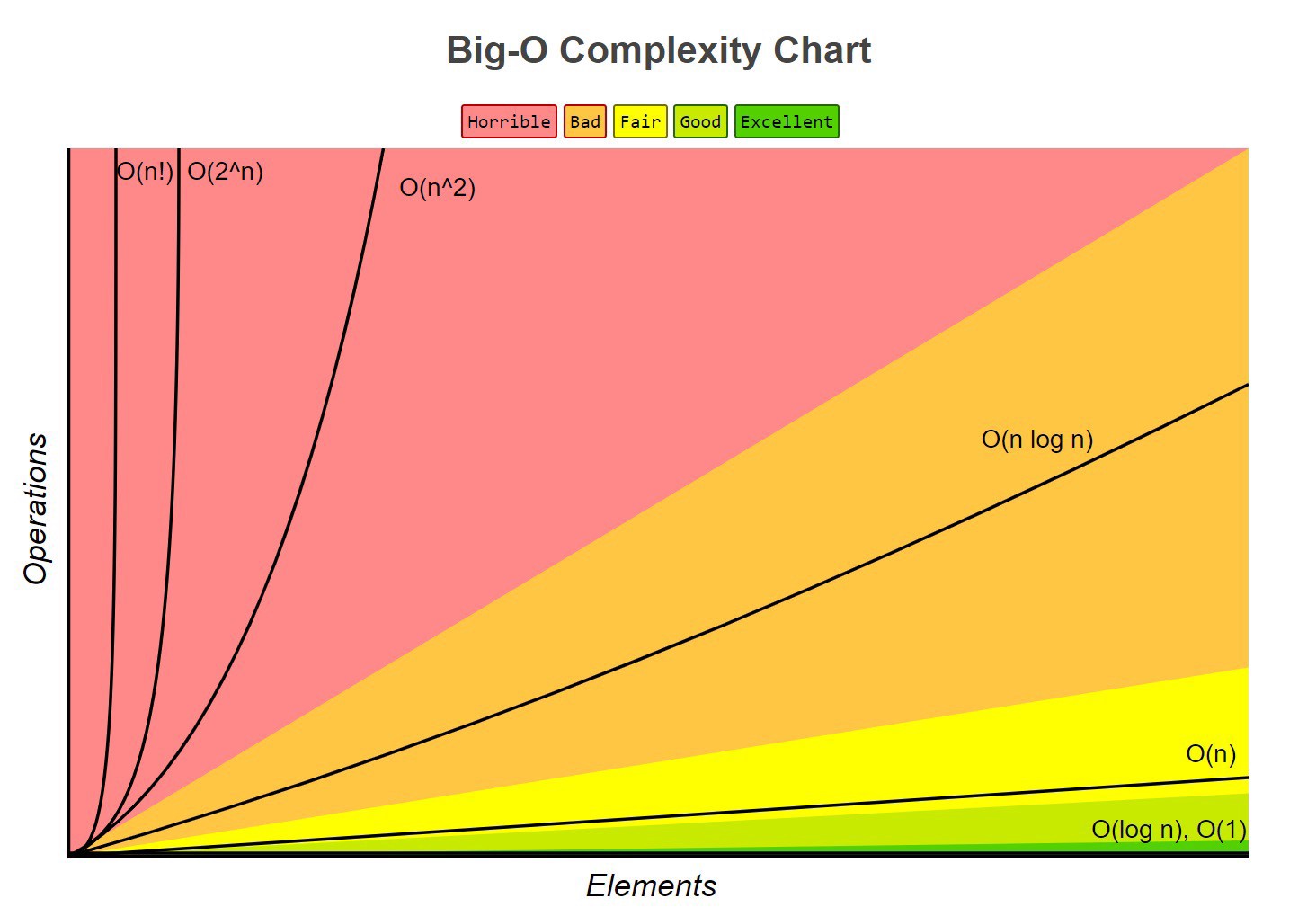
Mejor de los casos, promedio, peor de los casos.

Nada más importa el término de mayor tamaño.

**Clases de complejidad algorítmica**

Existen distintos tipos de complejidad algorítmica:

* **O(1) Constante:** no importa la cantidad de input que reciba, siempre demorara el **mismo tiempo**.
* **O(n) Lineal:** la complejidad crecerá de forma **proporcional** a medida que crezca el input.
* **O(log n) Logarítmica:** nuestra función crecerá de forma **logarítmica** con respecto al input. Esto significa que en un inicio crecerá rápido, pero luego se estabilizara.
* **O(n log n) Log lineal:** crecerá de forma **logarítmica** pero junto con una **constante**.
* **O(n²) Polinomial:** crecen de forma cuadrática. No son recomendables a menos que el input de datos en pequeño.
* **O(2^n) Exponencial:** crecerá de forma **exponencial**, por lo que la carga es muy alta. Para nada recomendable en ningún caso, solo para análisis conceptual.
* **O(n!) Factorial:** crece de forma **factorial**, por lo que al igual que el exponencial su carga es muy alta, por lo que jamas utilizar algoritmos de este tipo.



**Búsqueda lineal**

La **búsqueda lineal** es un algoritmo muy sencillo. Consta en buscar si un elemento se encuentra dentro de una lista, array o un sistema ordenado o no ordenado para poder determinar si el elemento se encuentra en el o forma parte de él.

¿Cuál es el peor caso del siguiente código? Si nos fijamos existe un **for loop** que crece según el tamaño de la lista, por lo cual nuestro Big O es O(n).

**Búsqueda binaria**

La **búsqueda binaria** toma una estrategia llamada “Divide y conquista”, la cual consiste en dividir el problema en 2 en cada iteración. Este algoritmo asume que la lista se encuentra ordenada, por lo que es necesario realizar este pasó primero.

La **búsqueda binaria** es uno de los mejores algoritmos que se tienen hoy en día para búsqueda, ya que reduce significativamente el número de pasos, y así disminuyendo nuestro Big O.

**Ordenamiento de burbuja**

El primer algoritmo de ordenamiento que veremos es el **ordenamiento de burbuja**. Es un algoritmo que recorre repetidamente una lista que necesita ordenarse. Compara elementos adyacentes y los intercambia si están en el orden incorrecto. Este procedimiento se repite hasta que no se requiere más intercambios, lo que indica que la lista se encuentra ordenada.

La lista se recorre una vez por cada elemento de la lista, los va comparando y ordenando.

**Ordenamiento por inserción**

El ordenamiento por inserción es uno de los algoritmos más comunes que estudian  
los Científicos del Cómputo. Es intuitivo y fácil de implementar, pero es muy  
ineficiente para listas de gran tamaño.

Una de las características del ordenamiento por inserción es que ordena en “su  
lugar.” Es decir, no requiere memoria adicional para realizar el ordenamiento  
ya que simplemente modifican los valores en memoria.

La definición es simple:

Una lista es dividida entre una sublista ordenada y otra sublista desordenada.  
Al principio, la sublista ordenada contiene un solo elemento, por lo que por  
definición se encuentra ordenada.

A continuación se evalua el primer elemento dentro la sublista desordenada para  
que podamos insertarlo en el lugar correcto dentro de la lista ordenada.

La inserción se realiza al mover todos los elementos mayores al elemento que  
se está evaluando un lugar a la derecha.

Continua el proceso hasta que la sublista desordenada quede vacia y, por lo  
tanto, la lista se encontrará ordenada.

Veamos un ejemplo:

Imagina que tienes la siguiente lista de números:

7, 3, 2, 9, 8

Primero añadimos 7 a la sublista ordenada:

**7**, 3, 2, 9, 8

Ahora vemos el primer elemento de la sublista desordenada y lo guardamos en  
una variable para mantener el valor. A esa variable la llamaremos valor\_actual.  
Verificamos que 3 es menor que 7, por lo que movemos 7 un lugar a la derecha.

**7**, 7, 2, 9, 8 (valor\_actual=3)

3 es menor que 7, por lo que insertamos el valor en la primera posición.

**3**, **7**, 2, 9, 8

Ahora vemos el número 2. 2 es menor que 7 por lo que lo movemos un espacio a la  
derecha y hacemos lo mismo con 3.

**3**, **3**, **7**, 9, 8 (valor\_actual=2)

Ahora insertamos 2 en la primera posición.

**2**, **3**, **7**, 9, 8

9 es más grande que el valor más grande de nuestra sublista ordenada por lo que  
lo insertamos directamente en su posición.

**2**, **3**, **7**, **9**, 8

El último valor es 8. 9 es más grande que 8 por lo que lo movemos a la derecha:

**2**, **3**, **7**, **9**, 9 (valor\_actual=8)

8 es más grande que 7, por lo que procedemos a insertar nuestro valor\_actual.

**2**, **3**, **7**, **8**, **9**

Ahora la lista se encuentra ordenada y no quedan más elementos en la sublista  
desordenada.

Antes de ver la implementación en Python, trata de implementarlo por ti mismo  
y compártenos tu algoritmo en la sección de comentarios.

Esta es una forma de implementar el algoritmo anterior:

**Ordenamiento por mezcla**

El **ordenamiento por mezcla** creado por **John von Neumann** el cual aplica el concepto de “divide y conquista”. Primero divide una lista en partes iguales hasta que quedan sublistas de 1 o 0 elementos. Luego las recombina en forma ordenada.

**Ambientes virtuales**

Los **ambientes virtuales** permiten aislar el ambiente para poder instalar diversas versiones de paquetes. A partir de *python 3* se incluye en la librería estándar en el módulo **venv**. Ningún ingeniero profesional de Python trabaja sin ellos.

**Pip** permite descargar paquetes de terceros para utilizar en nuestro programa, también permite compartir nuestros paquetes con terceros y también podemos definir la versión del paquete que necesitamos.

Para crear un ambiente virtual primer crearemos el directorio para nuestro proyecto.

*env*: ambiente virtual, siempre instalar dentro del ambiente virtual.

**¿Por qué graficar?**

Es importante que podamos traducir los datos que nos arrojan nuestro programa en un elemento visual, así podemos realizar reconocimientos de patrones, predicción de series, simplifica la interpretación y la conclusión acerca de los datos.

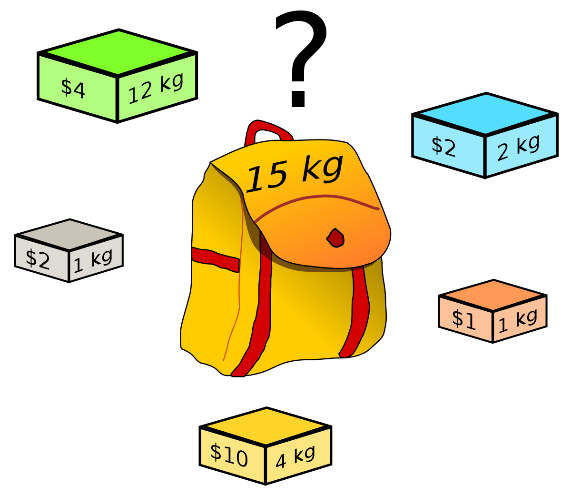
**Introducción a la optimización**

El concepto de **optimización** permite resolver muchos problemas de manera computacional. Cuando pensamos en un algoritmo de optimización debemos definir una función objetivo que debemos maximizar o minimizar, respetando una serie de limitantes que definamos.

**El problema del morral**

Imagina que eres un ladrón que entra a un museo pero tienes un gran problema, nada más tienes una mochila pero hay muchísimas cosas más de las que puedes cargar, por lo cual debes determinar cuáles artículos puedes cargar y te entregaran el mayor valor posible.

Para este problema sabemos que no podemos dividir los artículos, por lo que nuestra primera aproximación será evaluar los artículos.



Conclusiones

● Los tipos abstractos (clases) permiten crear programas poderosos que modelan al mundo.

● Podemos medir la eficiencia de diversos algoritmos.

● Las gráficas nos permiten encontrar patrones rápidamente.

● Optimización

**Algunos sitios donde encontrar retos:**

* [https://www.hackerrank.com](https://www.hackerrank.com/)
* <https://exercism.io/>
* <https://www.freecodecamp.org/learn>
* <https://www.codewars.com/dashboard>